Programs Constructs and Algorithm for Problem Solving

1. Program Constructs

Program constructs are fundamental building blocks used in programming to create logic, control flow, and structure within a program. They include:

1. **Sequential Execution**: Program statements are executed one after the other in the order they appear.
2. **Selection Constructs**:
   - **Conditional Statements (if-else)**: These allow the program to execute different blocks of code based on specified conditions.
   - **Switch Statements**: These provide a way to select one of many code blocks to be executed.
3. **Iteration Constructs**:
   - **Loops (for, while, do-while)**: These allow executing a block of code repeatedly until a certain condition is met.
   - **Iterators (foreach)**: These allow iterating over elements of collections or arrays.
4. **Modularization Constructs**:
   - **Functions/Methods**: These allow grouping a set of statements into a single unit and can be called multiple times from different parts of the program.
   - **Classes and Objects**: These allow organizing related functions and data into a single unit, providing encapsulation, inheritance, and polymorphism.

2. Variables, Expressions, and Statements

1. **Variables**: it is containers used to store data values.

   String firstName = "Java"
2. **Expression**: An expression is a combination of values, variables, operators, and function calls that results in a single value. Expressions can be simple or complex.

   Ex. result = 5 + 3 * 2

3. **Statement**: A statement is a complete line of code that performs an action or represents a command. Statements can be simple or compound (containing multiple smaller statements).

Assginment Statement

x = 10

Conditional Statement

```
if x > 5{
   print("x is greater than 5")
}else{
   print("x is not greater than 5")
}
```

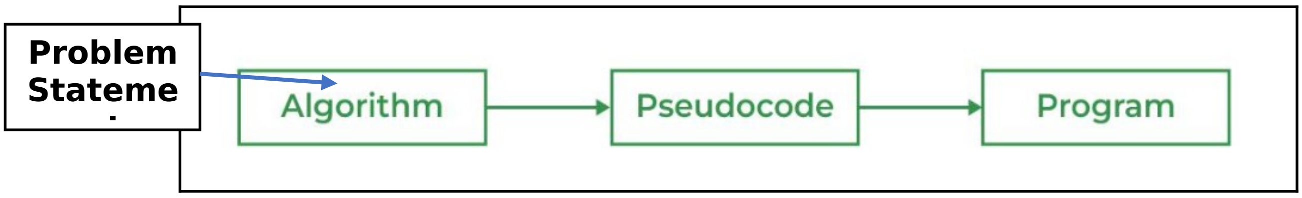
Loop Statement

```
for (int I =1; i<= 10; i++ {
  print(i)
}
```

3. Introduction to Pseudo Code:

Pseudocode *is defined as a step-by-step description of an algorithm,* is a way of **representing algorithms in a human-readable and high-level manner,** without being tied to any specific programming language syntax.

Pseudocode is a way to describe an algorithm's steps without using a specific programming language. It's a combination of programming language conventions and informal notation



Algorithm vs Pseudocode

Algorithm	Pseudocode
An Algorithm is used to provide a solution to a particular problem in form of a well-defined step-based.	A Pseudocode is a step-by-step description of an algorithm in code-like structure using plain English text.

Algorithm	Pseudocode
An algorithm only uses simple English words	Pseudocode also uses reserved keywords like if-else, for, while, etc.
These are a sequence of steps of a solution to a problem	These are fake codes as the word pseudo means fake, using code like structure and plain English text
There are no rules to writing algorithms	There are certain rules for writing pseudocode
Algorithms can be considered pseudocode	Pseudocode cannot be considered an algorithm
It is difficult to understand and interpret	It is easy to understand and interpret

Factorial of given Number:

Algorithm:

1. Start
2. Initialize a variable **factorial** to 1
3. Input a number **n**
4. If **n** is less than 0, display "Factorial is not defined for negative numbers."
5. If **n** is 0 or 1, set **factorial** to 1
6. For **i** from 2 to **n**:
   - Multiply **factorial** by **i**
7. Display **factorial**
8. End

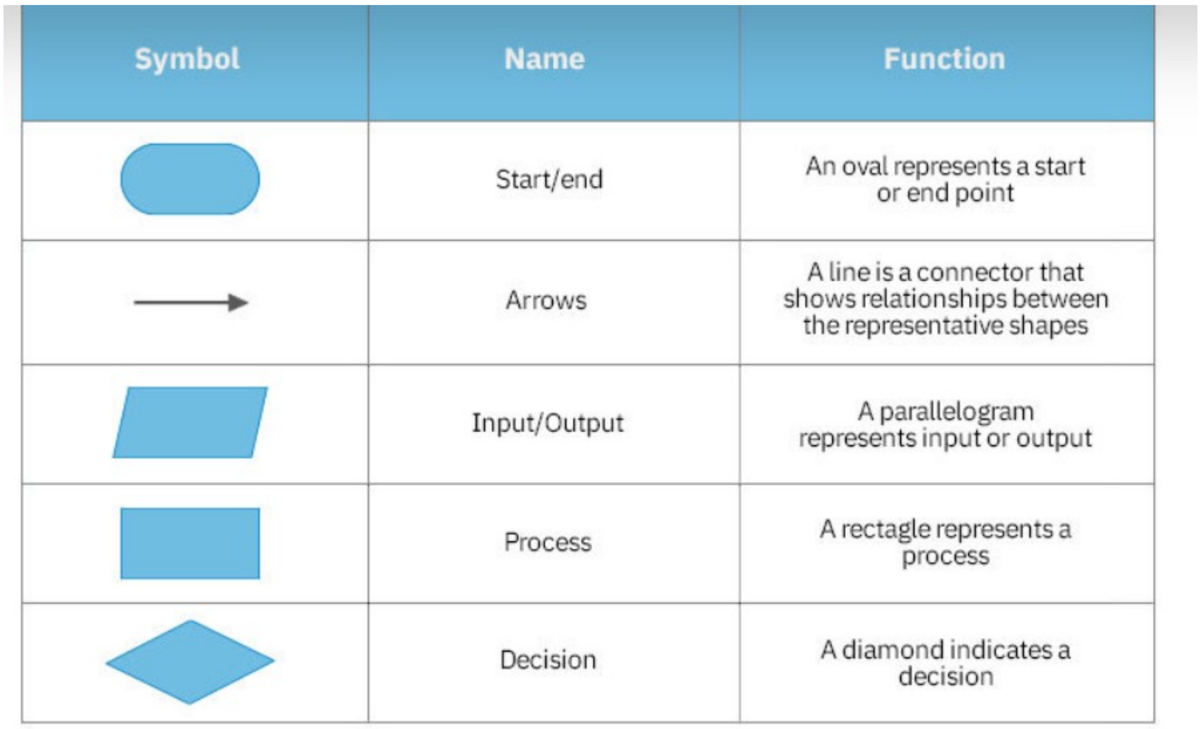
Pseudo Code:

```
BEGIN
   factorial = 1
   INPUT n
   IF n < 0 THEN
      DISPLAY "Factorial is not defined for negative numbers."
   ELSE IF n = 0 OR n = 1 THEN
      factorial = 1
   ELSE
      FOR i FROM 2 TO n DO
         factorial = factorial * i
```

```
        END FOR
    END IF
    DISPLAY factorial
END
```

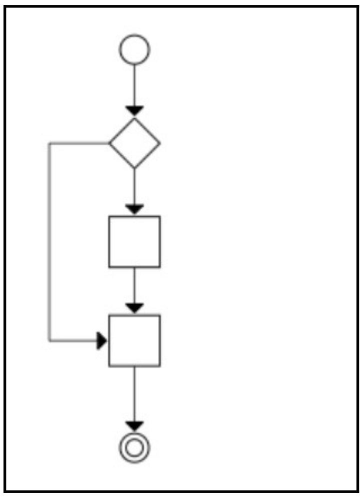
*A **Pseudocode** is defined as a step-by-step description of an algorithm. Pseudocode does not use any programming language in its representation instead it uses the simple English language text as it is intended for human understanding rather than machine reading.*

4. Control Flow Chart:



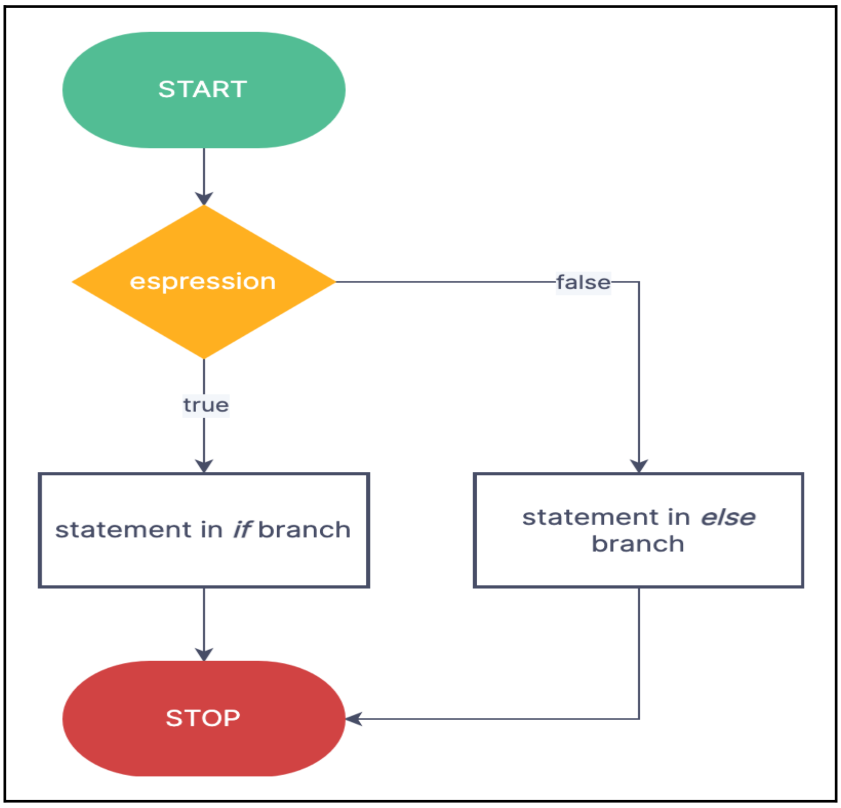
Symbol	Name	Function
⬭	Start/end	An oval represents a start or end point
→	Arrows	A line is a connector that shows relationships between the representative shapes
▱	Input/Output	A parallelogram represents input or output
▭	Process	A rectagle represents a process
◆	Decision	A diamond indicates a decision

<u>**If flow chart:**</u>



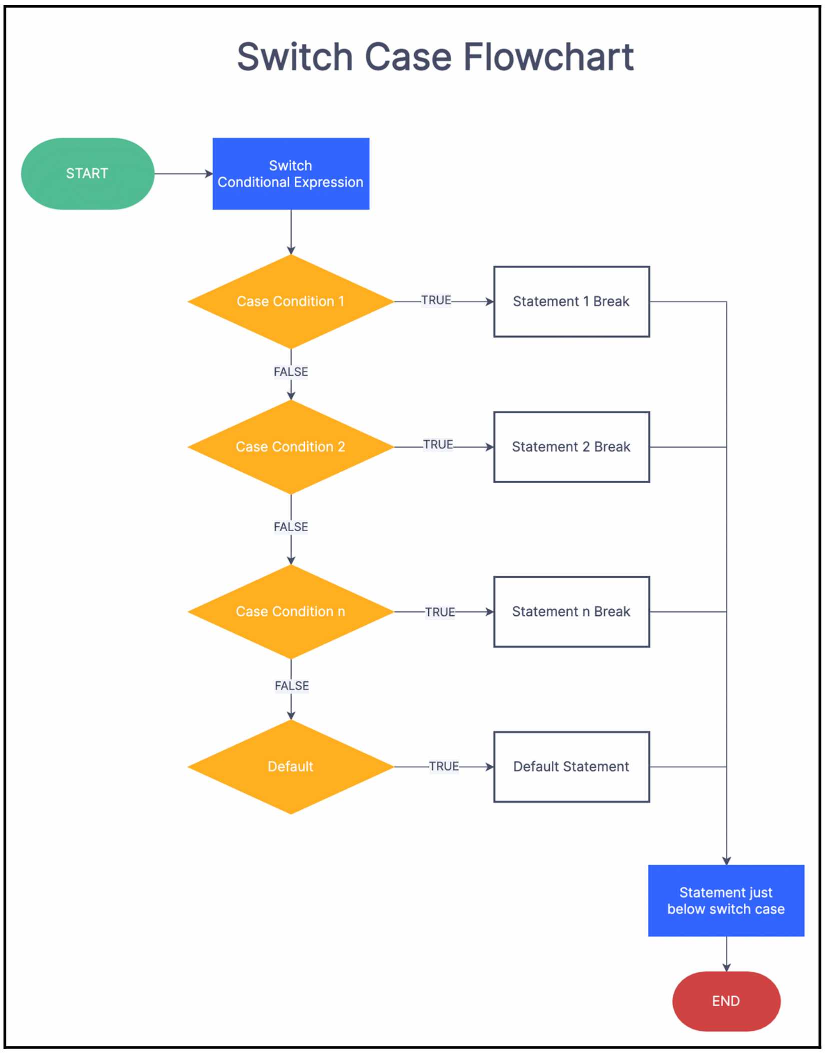
```
boolean a = ...;
if(a){
  print("A");
}
print("B");
```

If else flow chart

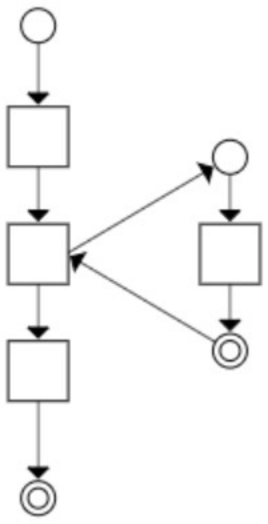


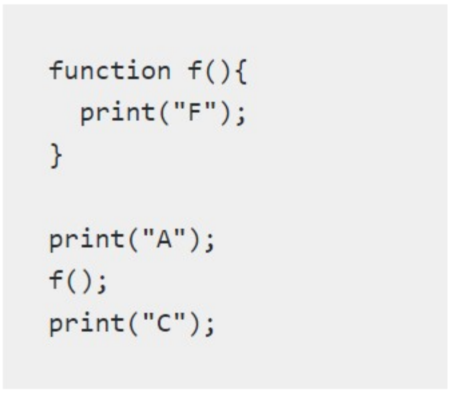
```
boolean a = ...;

if(a){
  print("A");
}else{
  print("B");
}

print("C");
```

Switch Case Flow Chart



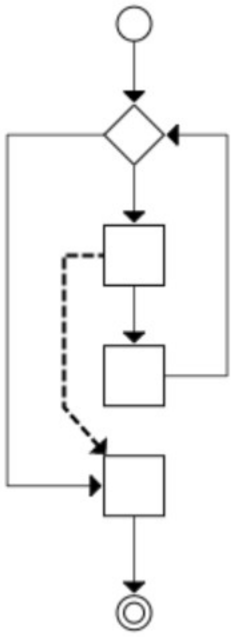
Function or Sub-Routine Flow Chart

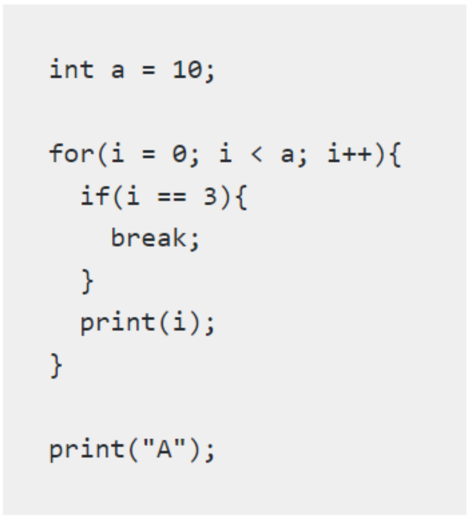




```
function f(){
  print("F");
}

print("A");
f();
print("C");
```

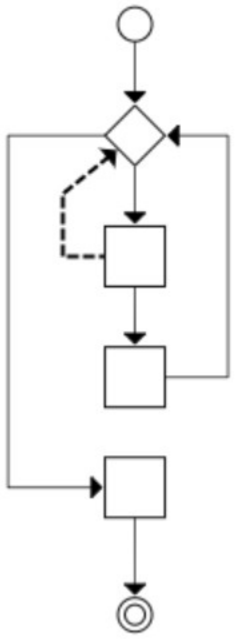
Break Flow Chart





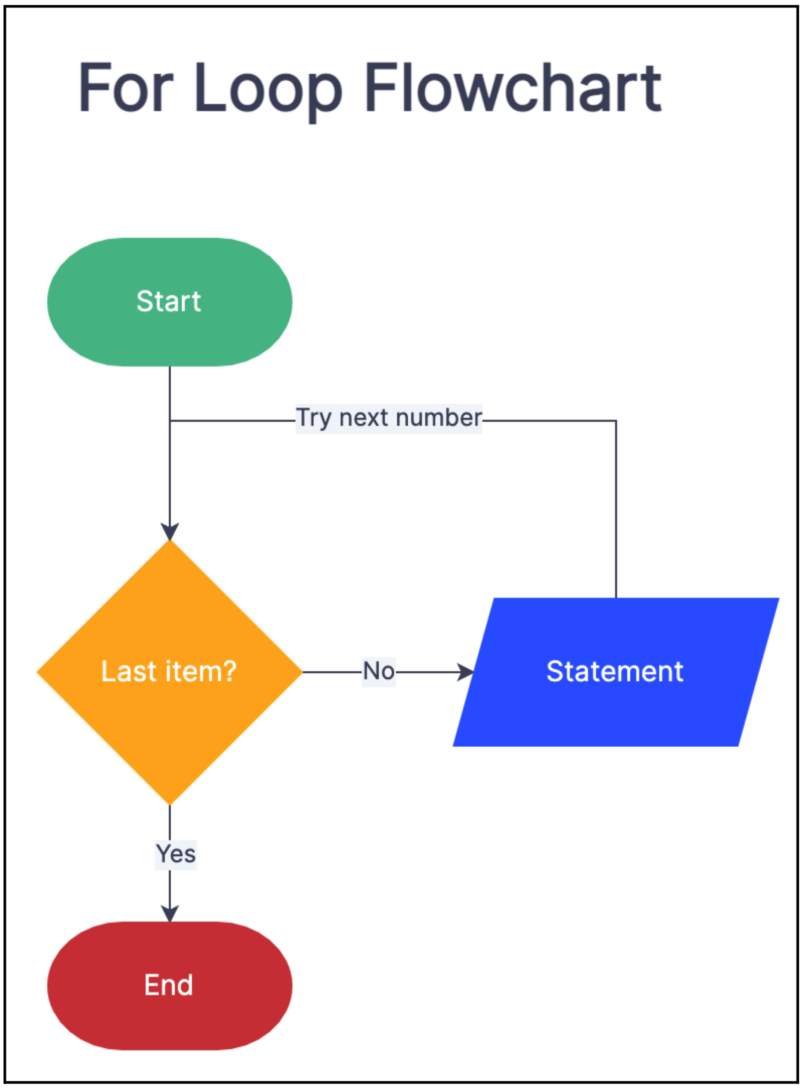
```
int a = 10;

for(i = 0; i < a; i++){
  if(i == 3){
    break;
  }
  print(i);
}

print("A");
```

Continue Flow Chart



```
int a = 3;

for(i = 0; i < a; i++){
  if(i == 1){
    continue;
  }
  print(i);
}

print("A");
```

For Loop Flowchart



5. Big O Notation

The Big O notation is the mathematical notation used to measure the algorithm’s efficiency in terms of time complexity or space complexity.
It is expressed as O(f(n)), where:

- "O" stands for "order of",
- "f(n)" represents a function of the input size "n".

For example:

- O(1): Constant time complexity. The algorithm's performance is independent of the input size.
- O(log n): Logarithmic time complexity. The algorithm's performance grows logarithmically with the input size.
- O(n): Linear time complexity. The algorithm's performance grows linearly with the input size.
- O(n^2): Quadratic time complexity. The algorithm's performance grows quadratically with the input size.
- O(2^n): Exponential time complexity. The algorithm's performance grows exponentially with the input size.

Time and Space Complexity

Algorithm	Time Complexity	Space Complexity
Linear Search	O(n)	O(1)
Binary Search	O(log n)	O(1)
Bubble Sort	O(n^2)	O(1)
Selection Sort	O(n^2)	O(1)
Insertion Sort	O(n^2)	O(1)
Merge Sort	O (n log n)	O(N)
Heap Sort	O(n log n)	O(1)
Quick Sort	O(n^2)	O(N)
Hashing		

Sources:
https://www.naukri.com/code360/library/time-and-space-complexities-of-sorting-algorithms-explained

6. Bubble Sort

Algorithm:

1. **Compare Adjacent Elements**:
    - Start from the first element and compare it with the next element.

    - If the first element is greater than the second element, swap them.
2. **Repeat the Process**:
    - Continue this process for each pair of adjacent elements in the array.
    - After the first pass, the largest element will be at the end of the array.
    - Repeat the process for the remaining elements until the entire array is sorted.

7. Selection Sort:

Algorithm:

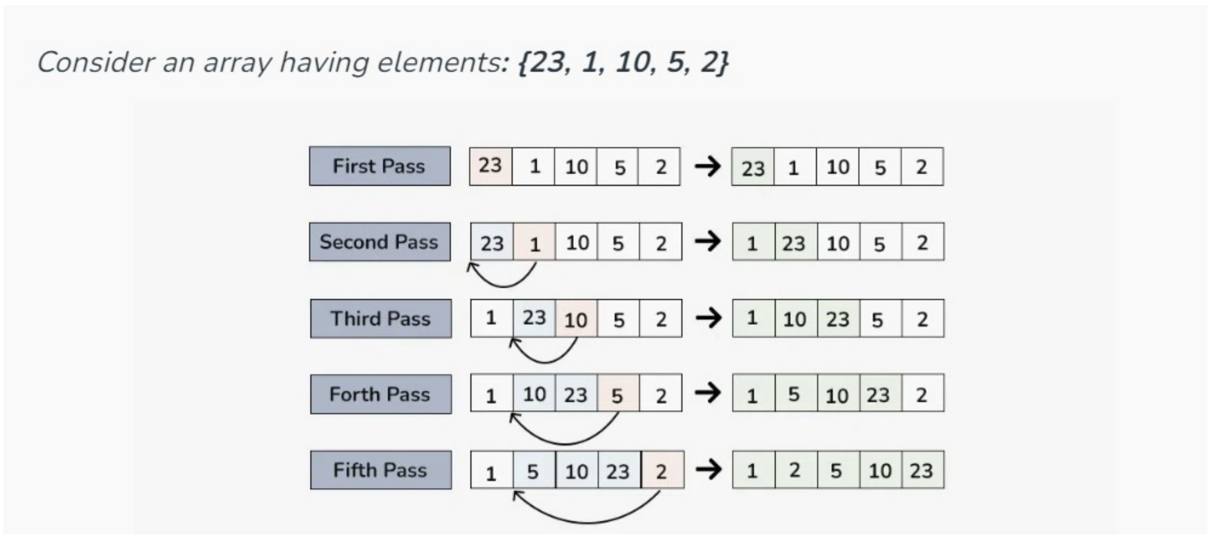
1. **Find the minimum element**:
    - Start from the first element and assume it is the minimum.
    - Compare the minimum with each of the subsequent elements in the array.
    - If you find an element smaller than the current minimum, update the minimum.
2. **Swap the minimum element with the first unsorted element**:
    - Once the minimum element is found, swap it with the first unsorted element.
3. **Repeat the process**:
    - Move the boundary of the sorted subarray one element to the right.
    - Repeat steps 1 and 2 for the remaining unsorted subarray until the entire array is sorted.

8. Insertion Sort

Algorithm:

- We have to start with second element of the array as first element in the array is assumed to be sorted.
- Compare second element with the first element and check if the second element is smaller, if so then swap them.
- Move to the third element and compare it with the second element, then the first element and swap as necessary to put it in the correct position among the first three elements.
- Continue this process, comparing each element with the ones before it and swapping as needed to place it in the correct position among the sorted elements.

- Repeat until the entire array is sorted.



9. Merge Sort

Algorithm:

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

10. Quick Sort

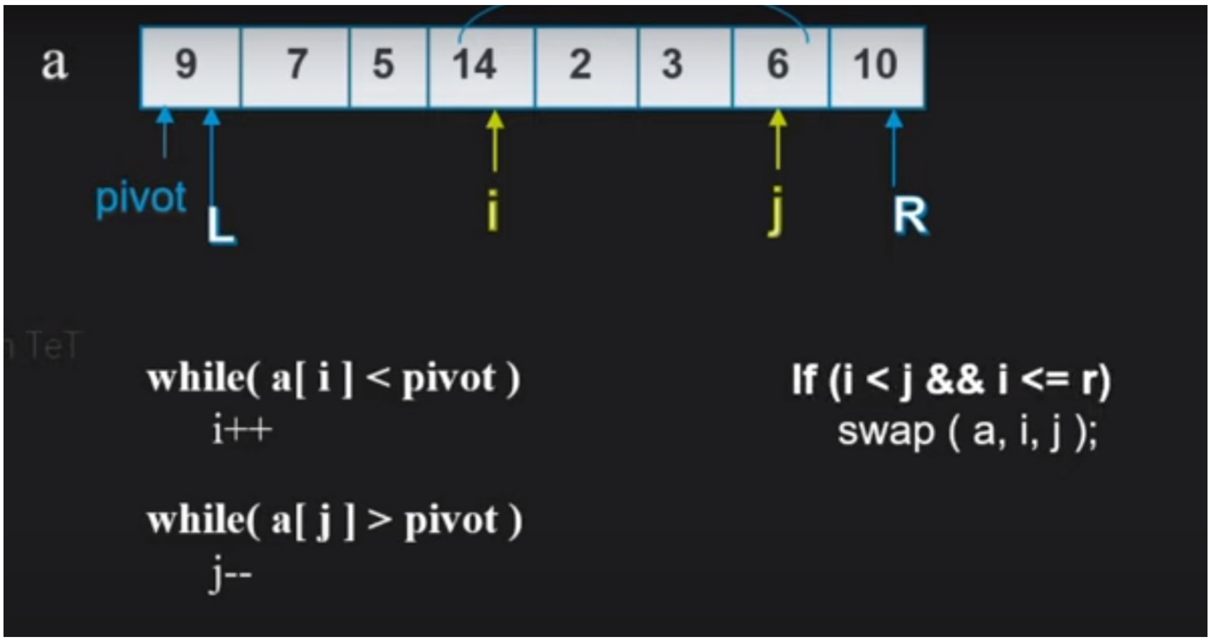
Algorithm:

1. **Choose a Pivot**:
    - Select a pivot element from the array. This can be done in various ways, such as selecting the first, last, or middle element of the array.
2. **Partitioning**:
    - Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it. After partitioning, the pivot is in its final sorted position.
3. **Recursively Sort the Sub-arrays**:

- Recursively apply the above steps to the sub-array of elements with smaller values and the sub-array of elements with greater values.

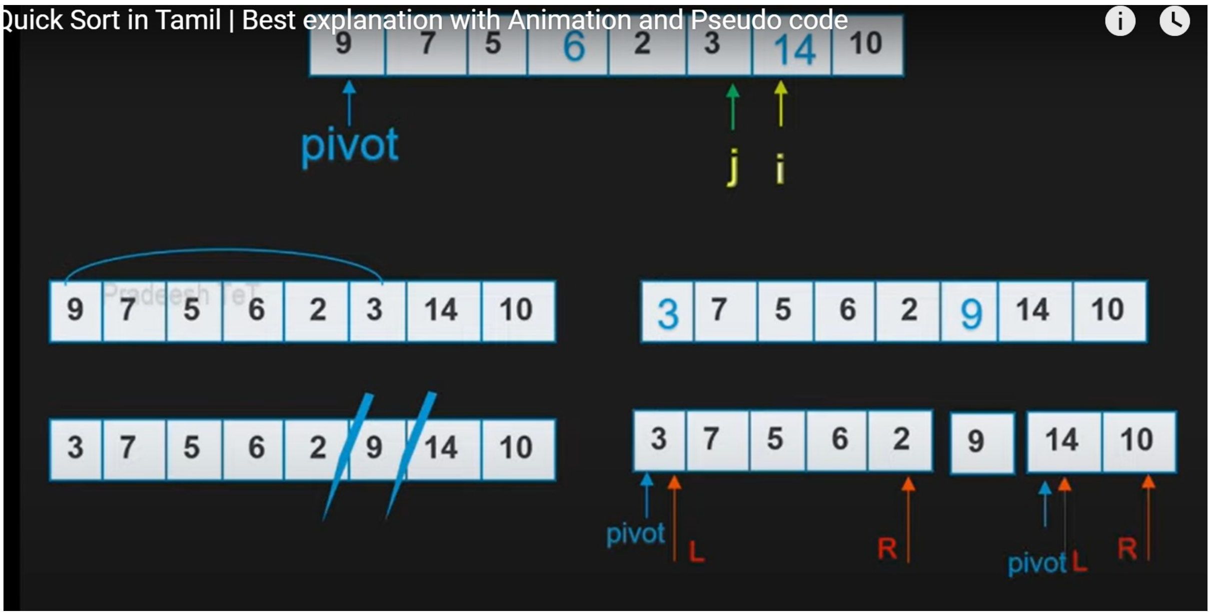
Rules:
When I and j is stopped, swap a[i] and a[j]
When I and j crossed over, swap a[j] with pivot

Int[] arr = new int[8] // index 0 to 7,   Left =0 , Right =7, I =1, j=7

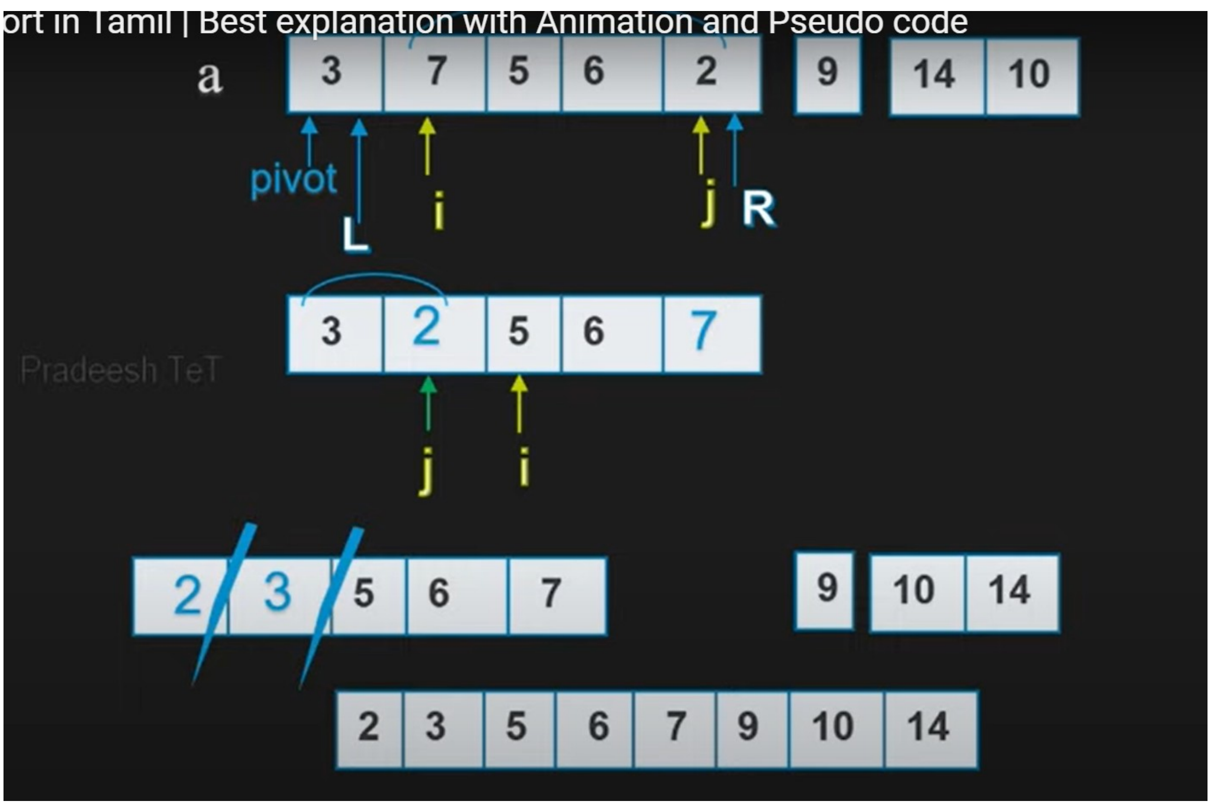


```
while( a[ i ] < pivot )
    i++

while( a[ j ] > pivot )
    j--
```

```
If (i < j && i <= r)
    swap ( a, i, j );
```



Quick Sort in Tamil | Best explanation with Animation and Pseudo code
pivot
j
i
Pradeesh TeT
pivot
L
R
pivot
L
R



ort in Tamil | Best explanation with Animation and Pseudo code
a
pivot
L
i
j
R
Pradeesh TeT
j
i

11. Heap Sort

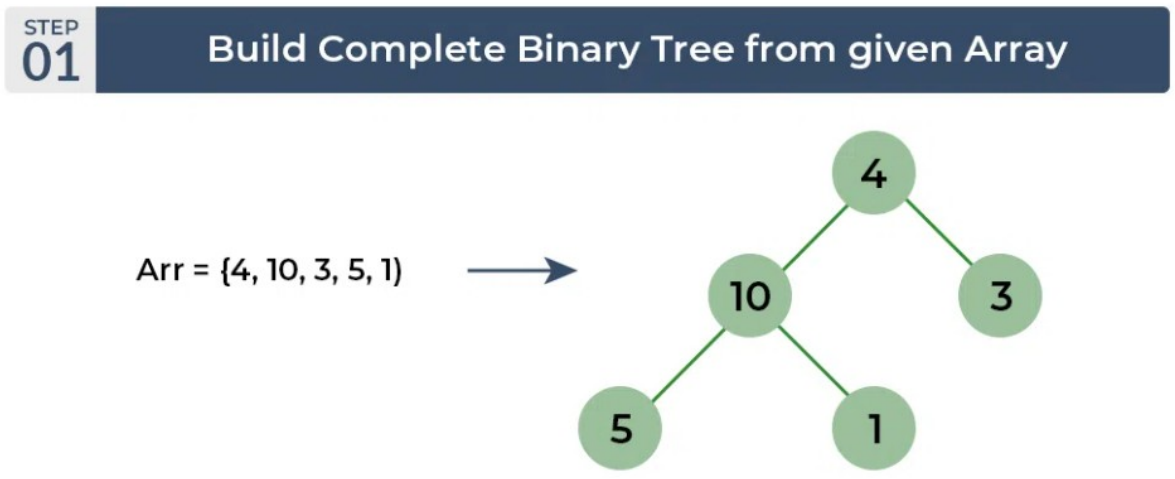
Algorithm:

*First convert the array into heap data structure using heapify, then one by one delete the root node of the **Max-heap** and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.*

- *Build a heap from the given input array.*
- *Repeat the following steps until the heap contains only one element:*
  - *Swap the root element of the heap (which is the largest element) with the last element of the heap.*
  - *Remove the last element of the heap (which is now in the correct position).*
  - *Heapify the remaining elements of the heap.*
- *The sorted array is obtained by reversing the order of the elements in the input array.*

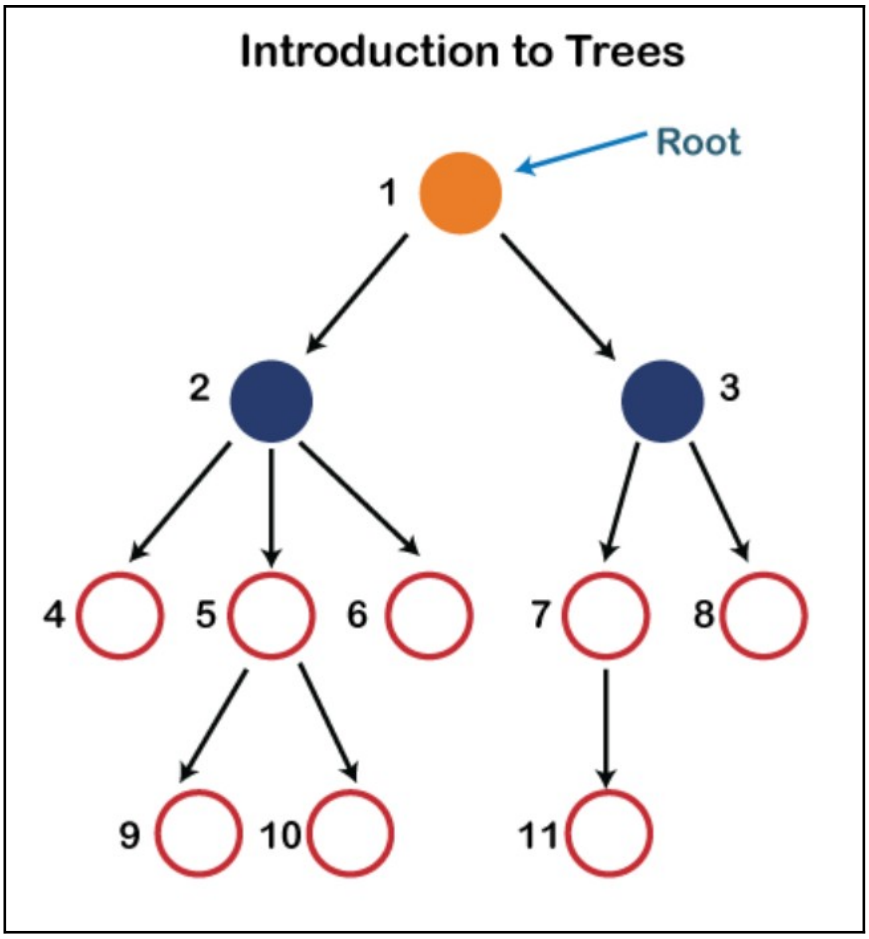
***Transform into max heap:** After that, the task is to construct a tree from that unsorted array and try to convert it into [max heap.]()*

- *To transform a heap into a max-heap, the parent node should always be greater than or equal to the child nodes*
  - *Here, in this example, as the parent node **4** is smaller than the child node **10,** thus, swap them to build a max-heap.*
- *Now, **4** as a parent is smaller than the child **5**, thus swap both of these again and the resulted heap and array should be like this:*



12. Trees:

A tree is also one of the data structures that represent hierarchical data.



- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.

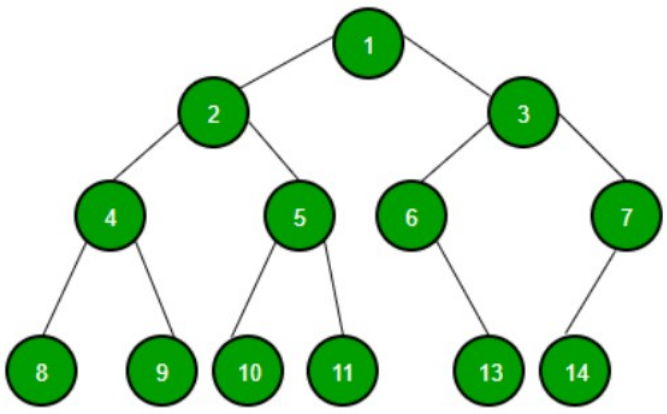
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has at least one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Different Type of Tress:

1. Binary Tree
2. Binary Search Tree
3. AVL Tree (*Adelson Velsky Lindas)*
4. B Tree
5. B+ Tree

13. Binary Tree

A **Binary Tree Data Structure** is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. It is commonly used in computer science for efficient storage and retrieval of data, with various operations such as insertion, deletion, and traversal.



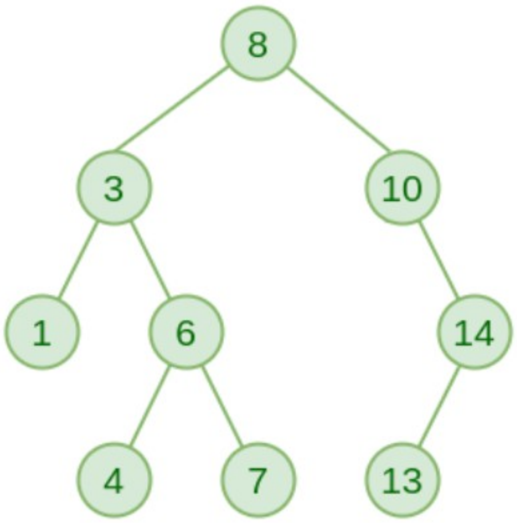
Traversal:

1. Preorder Traversal → Root, Left, Right → 1, 2, 4, 8, 9, 5, 10, 11, 3, 6, 13, 7, 14
2. In-order Traversal →Left, Root, Right → 8, 4, 9, 2, 10, 5, 11, 1, 6, 13, 3, 14, 7
3. Post Order Traversal → Left, Right, Root → 8, 9, 4, 10, 11, 5, 2, 13, 6, 14, 7, 3, 1

14. Binary Search Tree

Properties:

1. Each node can have at most 2 nodes, i.e ,0 , 1 and 2
2. The values of left node should be lesser than parent node
3. The values of right node should be greater than parent node



Binary Search Tree

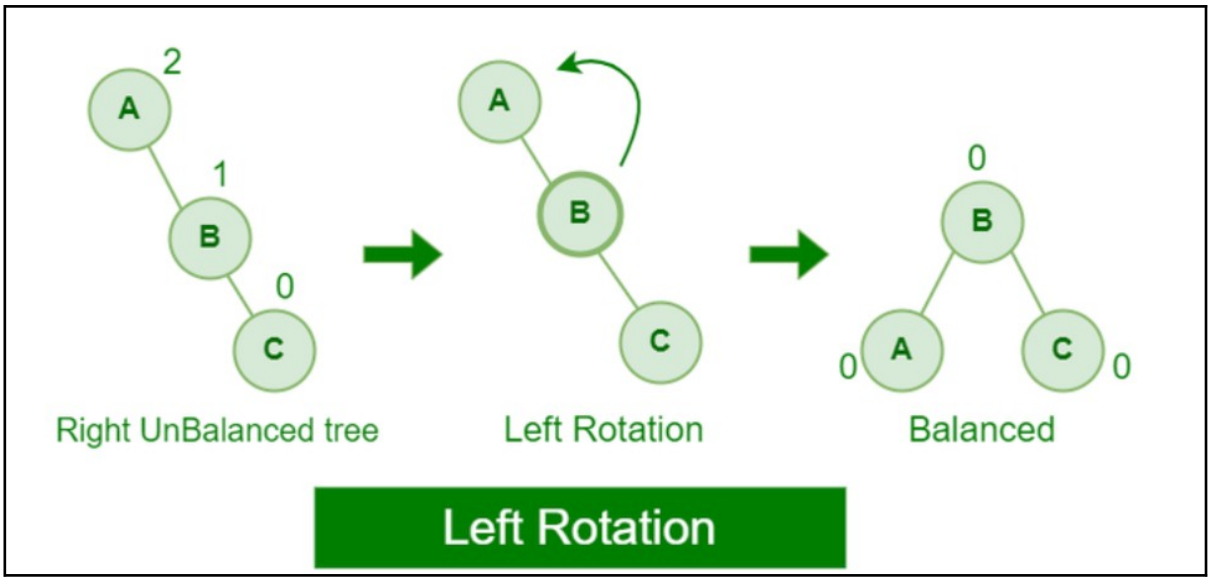
Draw Binary Search Diagram for following data
34, 27, 19, 51, 44, 30, 33, 60, 38, 63

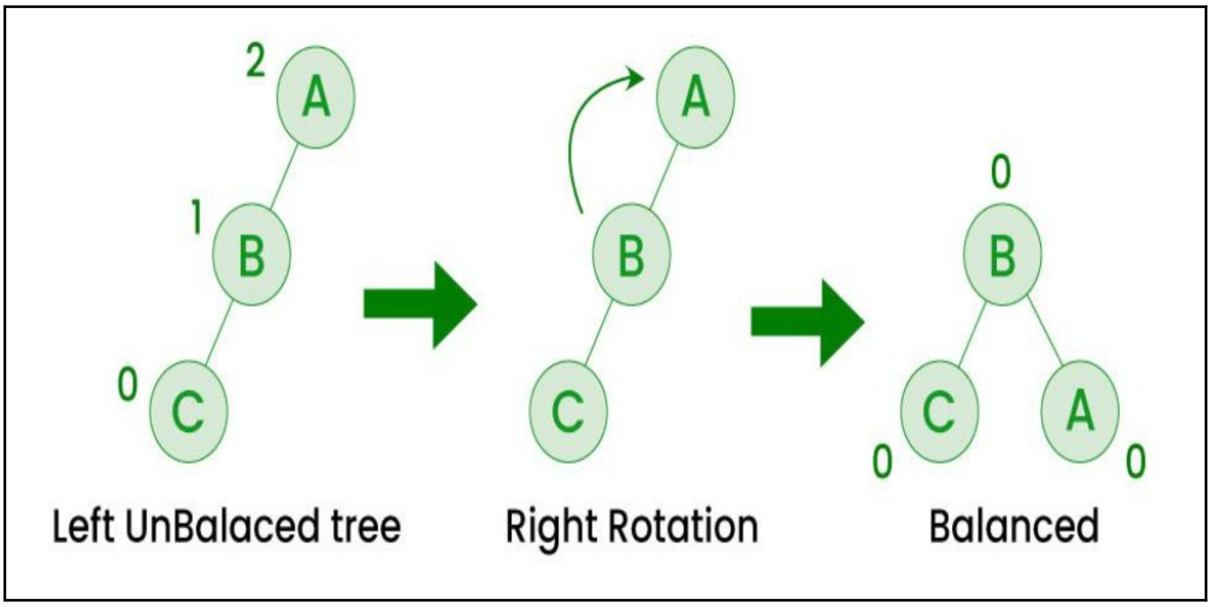
15. AVL Tree (Adelson Velsky Lindas)

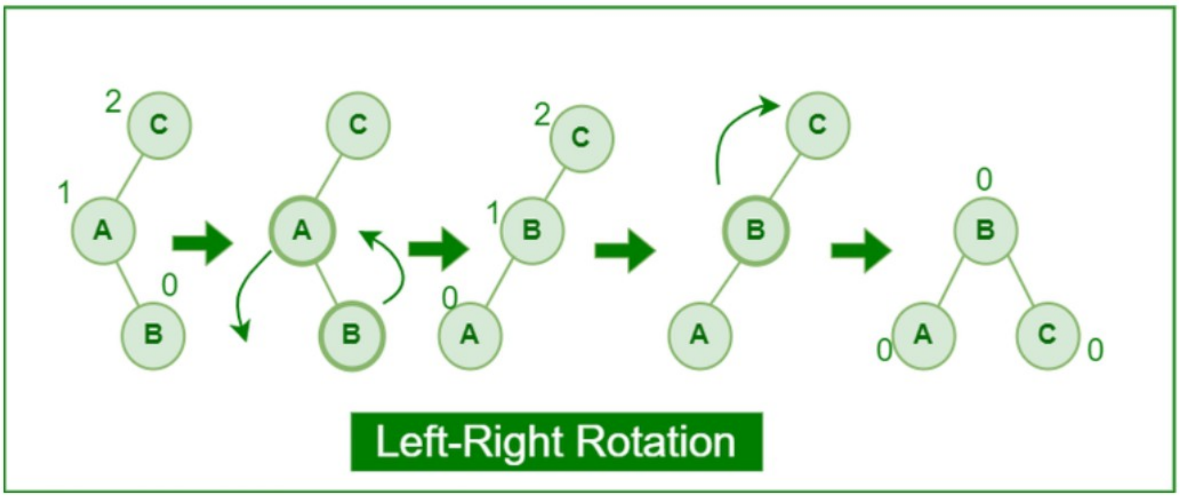
It is Self balancing binary tree *where the difference between heights of left and right subtrees for any node cannot be more than one.*
Balance factor can be either 0 or 1 or -1
Balance Factor can be evaluated via the formula
BF = Left child – right child
If you assessed Binary Tree is as In-balanced Binary Tree, convert it to Balanced Binary Tree.

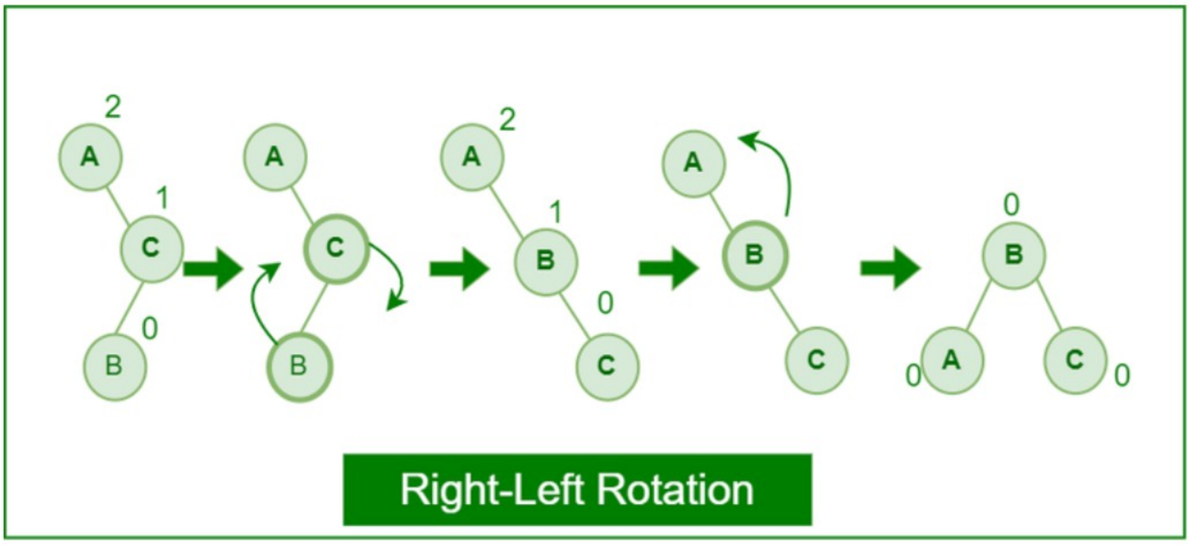
Ways to convert to Balanced Tree

1. Left Rotation
2. Right Rotation
3. Left Right Rotation
4. Right Left Rotation


2
A
1
B
0
C
Right UnBalanced tree
A
B
C
Left Rotation
0
B
0
A
C
0
Balanced
Left Rotation


2
A
1
B
0
C
Left UnBalaced tree
A
B
C
Right Rotation
0
B
0
C
A
0
Balanced


2
C
1
A
0
B
C
A
B
2
C
1
B
0
A
C
B
A
0
B
0
A
C
0
Left-Right Rotation



16. Graph

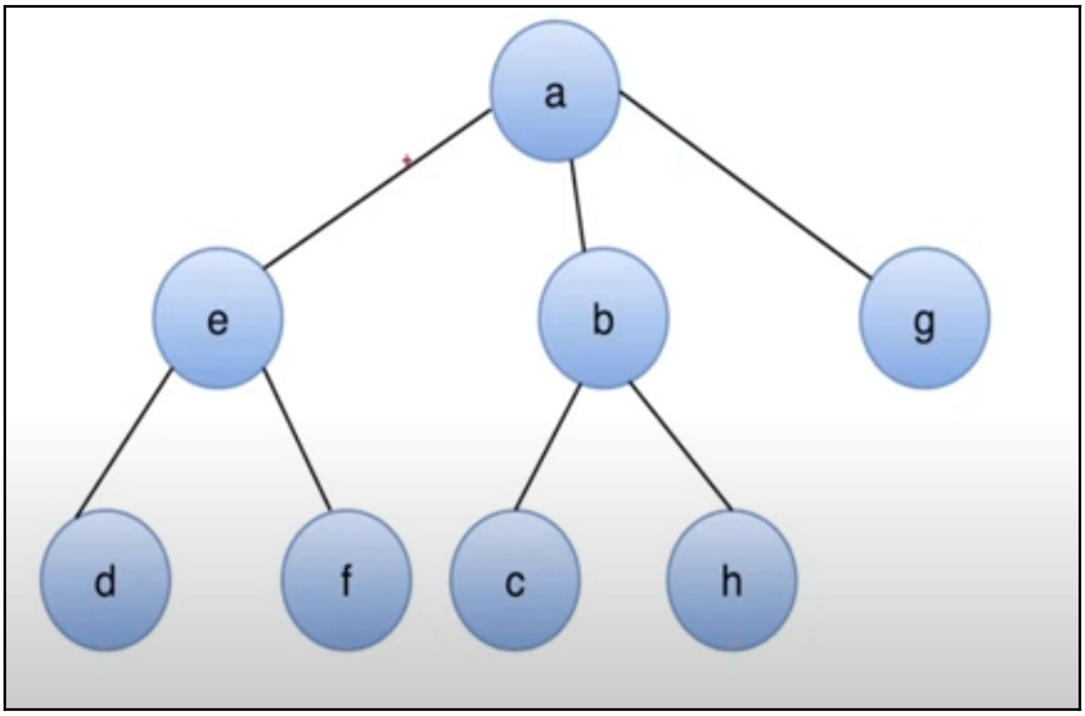
Graph is group of vertices(nodes) and edges. Vertices are connected by edges.
Eg. Social Media
Locations in the map.

Types of Graph:

1. BFS (Breadth First Search)
2. DFS (Depth First Search)

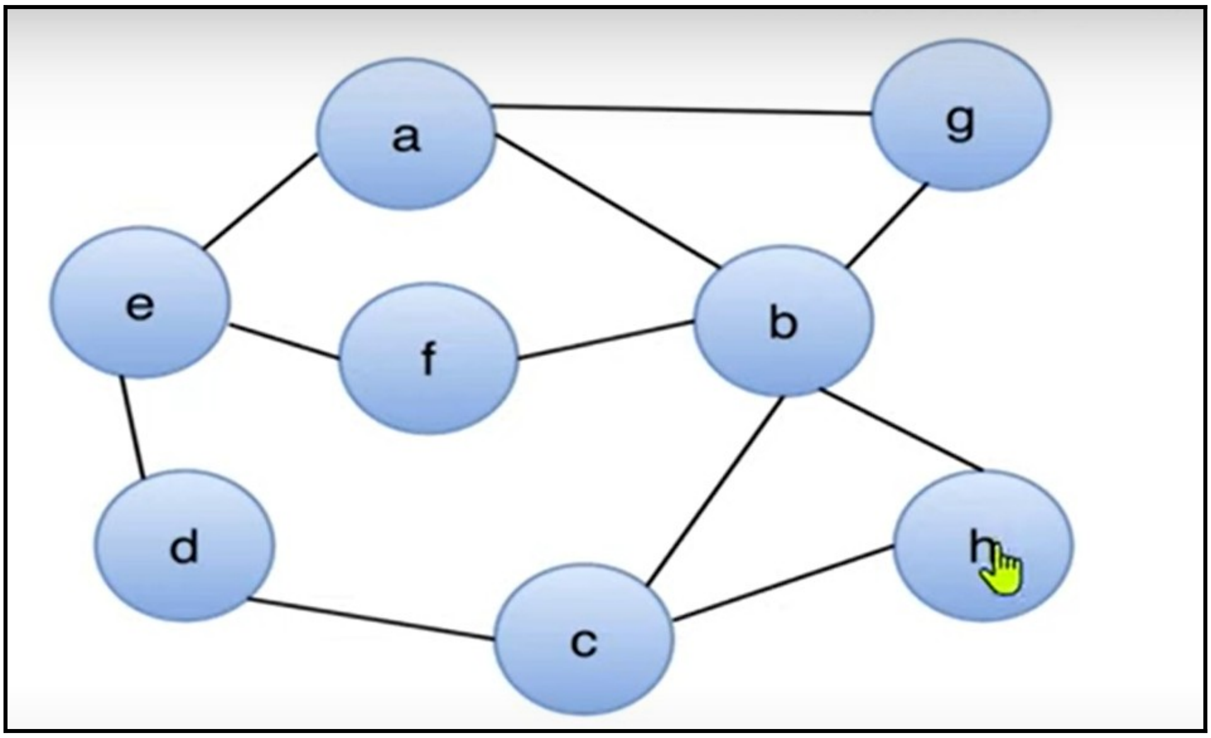
17. Breadth First Search

From Tree Structure



Traversal: a, e, b, g, d, f, c, h

From Graph



Traversal: a, e, b, g, f, d, c, h

18. Depth First Search

Tree Traversal: a, e, d, f, b, c, h, g

Graph Traversal: a, e, d, c, b, g, f, h

19.

20. Hashing

Hashing is a fundamental data structure that efficiently stores and retrieves data in a way that allows for quick access. It involves mapping data to a specific index in a hash table using a **hash function** that enabling fast retrieval of information based on its key. This method is commonly used in databases, **c**aching systems, and various programming applications to optimize search and retrieval operations.

Purpose:

With only one iteration you can find that element.

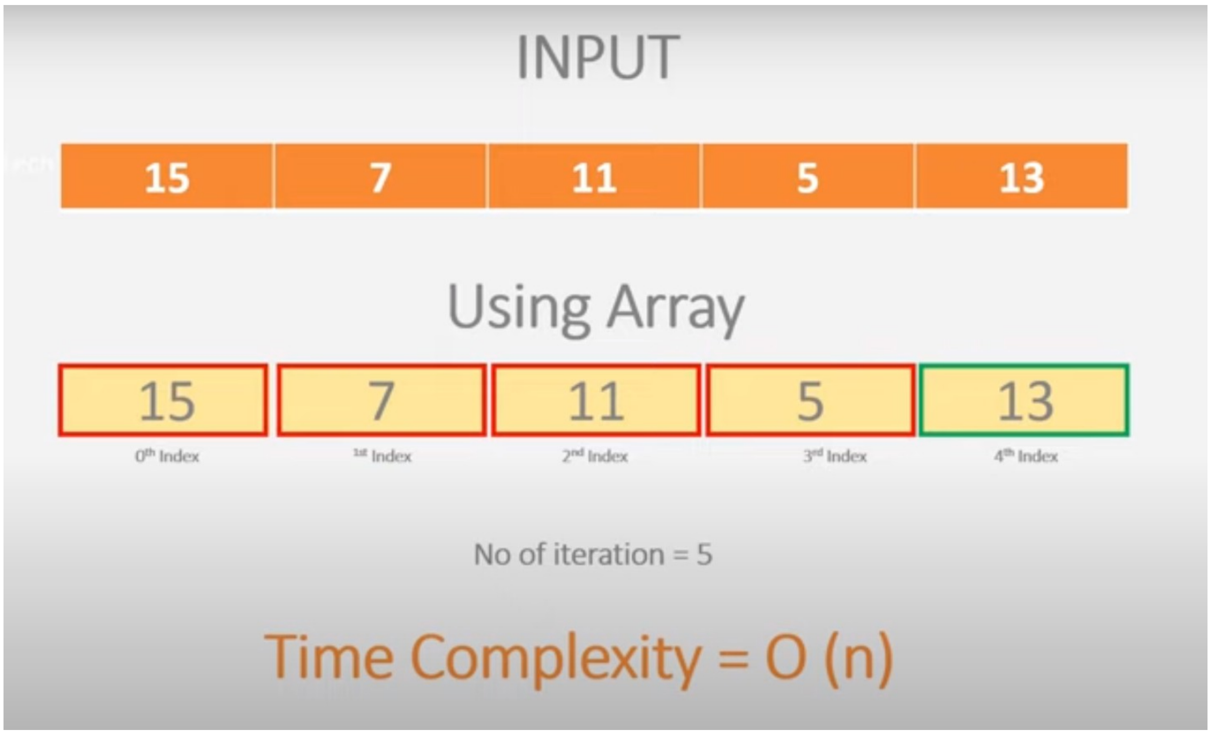
O(1)

Hashing function

The **hash function** is a function that takes a **key** and returns an **index** into the **hash table**. The goal of a hash function is to distribute keys evenly across the hash table, minimizing collisions (when two keys map to the same index).

Common hash functions include:

- **Division Method:** Key % Hash Table Size
- **Multiplication Method:** (Key * Constant) % Hash Table Size
- **Universal Hashing:** A family of hash functions designed to minimize collisions



INPUT

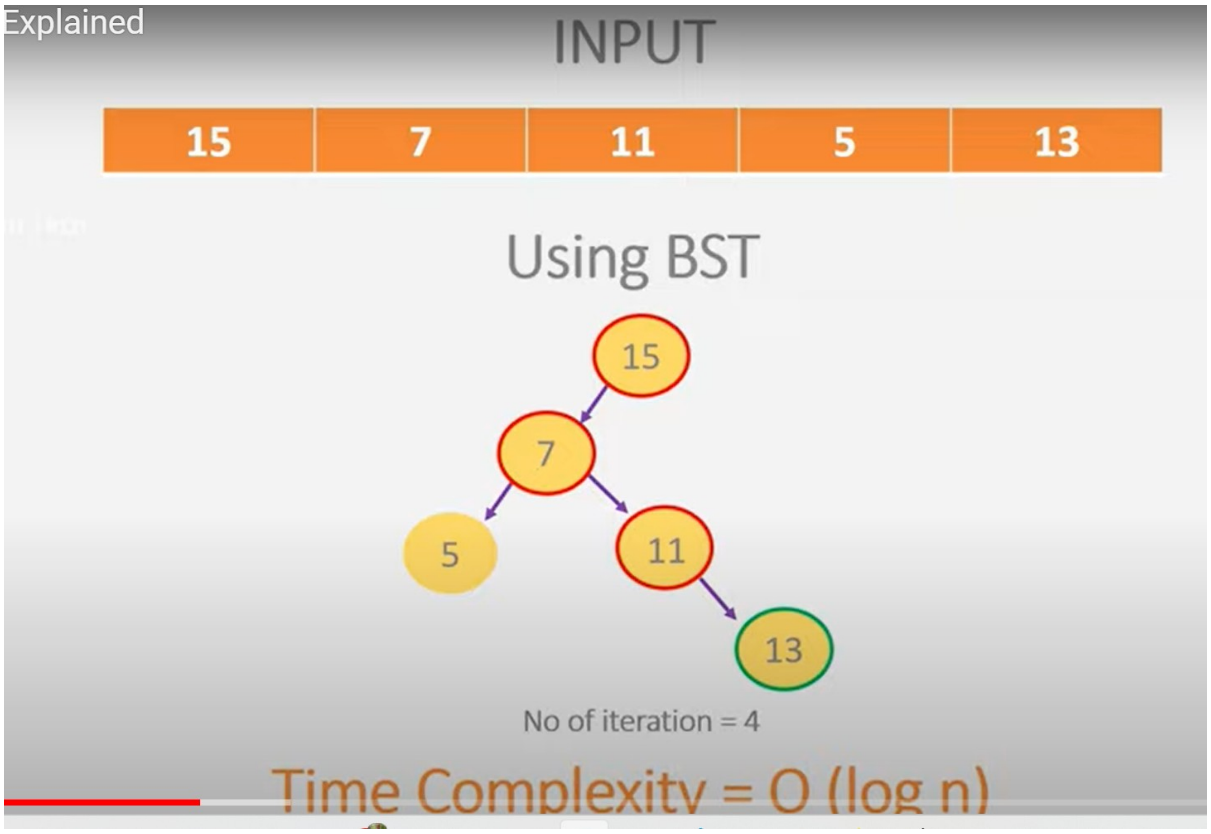
15	7	11	5	13

Using Array

15	7	11	5	13
0th Index	1st Index	2nd Index	3rd Index	4th Index

No of iteration = 5

Time Complexity = O (n)



Explained

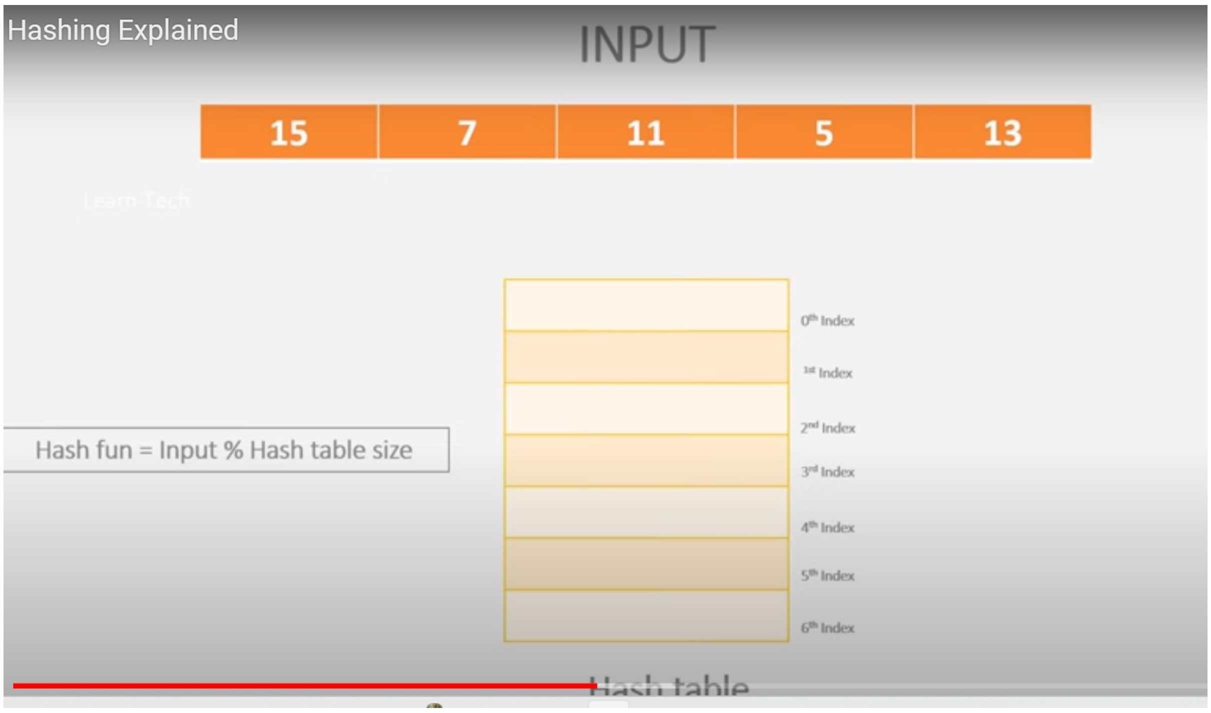
INPUT

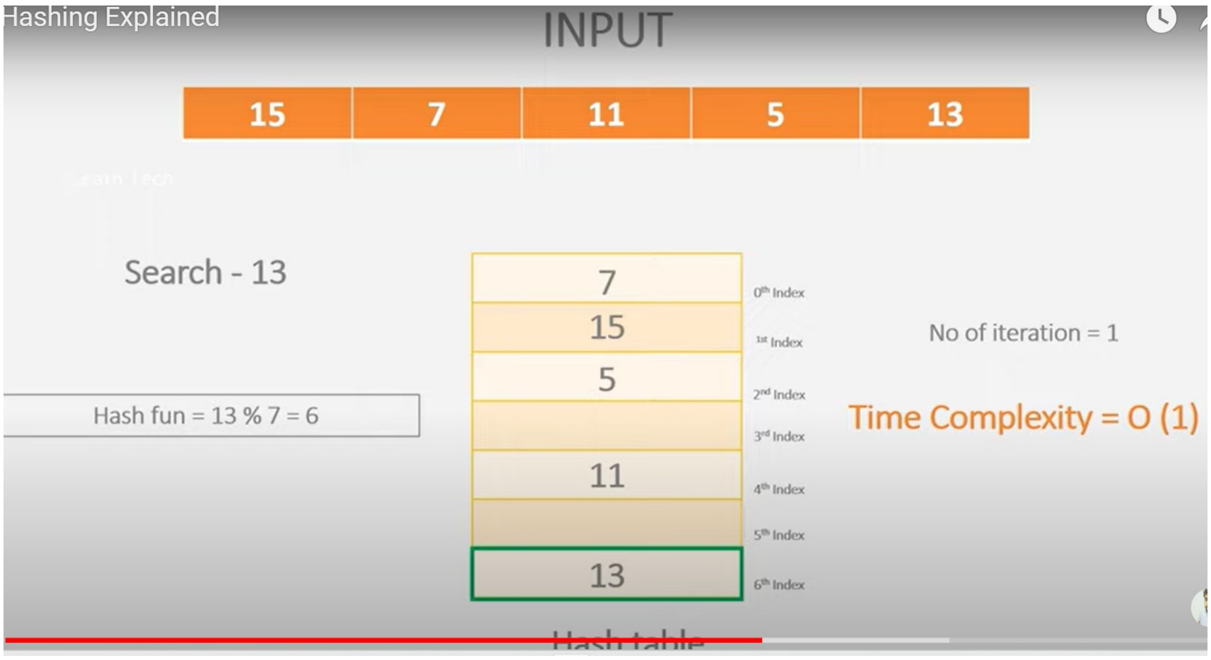
15	7	11	5	13

Using BST

No of iteration = 4

Time Complexity = O (log n)





Data Collision:

3. Separate Chaining – linked list concept
4. Open Addressing – 3 ways to handle open addressing
5. Close Addressing -
    -

21. Java Program

1. Control structure
2. Array
3. String
4. OOPs

1. Look coding MCQ question – 10 question each
2. Write a program for Linear and Binary Search.

X = a+b/d*e-f(a-b)/c;

A++ - post
++A - pre

10 < 45 -> result would either true or false

If Loop

1. Simple If loop
2. If else loop
3. If elseif loop
4. If ladder (if with if)

22. Brute Force Algorithm

A brute force algorithm is a simple method for solving a problem by trying every possible solution, rather than using advanced techniques to improve efficiency.

Different Algorithms in Brute Force Algorithm

1. String Matching
2. Closest Pair
3. Conver Hull
4. Assignment
5. TSP
6. Knapsack

Path
Classpath